Université Pierre & Marie Curie

Rapport de stage - juillet-août 2006

Paramétrage d'agents dans le cadre de la navigation coordonnée pour les jeux vidéo

---

Guillaume Riby

Responsable de stage: Olivier Sigaud

Master Informatique Spécialité STL
Année 2005 / 2006

Table des matières

Préface

Partie 1. Introduction et problématique
- A. Description des objectifs
- B. Apports scientifiques

Partie 2. Environnement de développement : Kodamat
- A. Le projet Kodabot
- B. SATOM et SATOMJAVA
- C. SMUGLU

Partie 3. La navigation dans Kodabot
- A. Navigation *StraightToTarget*
- B. Le principe de GACS
- C. Navigation GACS++

Partie 4. VisioGacs - *Application de paramétrage en ligne*
- A. Motivations
- B. Mise en place
- C. Structure de l'application

Partie 5. Paramétrage et résultats obtenus
- A. Force vers le but et contournement d'obstacles
- B. Evitement des autres entités
- C. Maintien de formation

Conclusion

Lexique

Annexes
- A. Manuel d'utilisateur de VisioGacs
- B. Capture de vidéos dans Counter-Strike

Préface

Dans le cadre de ma formation de Master spécialité informatique, à l'Université des Sciences et Techniques de Paris 6, il m'a été proposé de suivre un cursus dit de Magistère, ce cursus impose entre autre la réalisation d'un stage à la sortie de la première année.
L'année passée, j'ai effectué avec deux autres étudiants un projet intitulé « Module de navigation coordonnée dans le cadre des jeux vidéo » directement en lien avec le Laboratoire d'Informatique de Paris 6 et plus précisément l'AnimatLab. Durant ce projet qui a duré tout un semestre, j'ai appris à utiliser l'environnement de développement présent, c'est-à-dire, les différents frameworks de composants ainsi que les méthodes de travail à adopter, ce qui dans un premier temps n'a pas été évident. Renforcé par l'expérience acquise durant ce projet et intéressé par le jeu vidéo, c'est tout naturellement que je me suis tourné vers l'AnimatLab pour entreprendre ce stage.
Mon responsable de stage est Olivier Sigaud que je remercie pour m'avoir accepté durant les mois de juillet et août. La plupart du temps, j'ai été encadré par Thomas Degris qui effectue actuellement sa thèse concernant l'apprentissage par renforcement. Mais j'ai aussi travaillé aux côtés d'autres étudiants stagiaires tout comme moi, étudiants de différentes filières aussi bien de parcours d'intelligence artificielle que de génie logiciel.

Dans un premier temps j'introduirai le sujet et les problèmes posés. J'en profiterai pour faire une brêve description du principe de *GACS*, l'algorithme de navigation développé dans la thèse de Fabien Flacher.
Ce stage s'incrivant dans la continuité du projet effectué précédemment décrit, je vais donc éviter tant que possible de donner trop de détails concernant l'environnement de développement bien que quelques explications sommaires soient indispensables.
Puis je poursuivrai sur la représentation actuelle de la partie Kodabot, c'est-à-dire, les différents constituants de l'agent comme la planification et les différents types de navigation.
Une fois ces bases posées je pourrai présenter l'application que j'ai mise en place, application de paramétrage en ligne nommée VisioGacs; enfin j'exposerai les résultats obtenus grâce à cette application mais aussi les problèmes non résolus.

Partie 1. Introduction et problématique

Ce projet s'inscrit dans la continuité du projet annuel de première année de Master que j'avais effectué en trinôme quelques mois plus tôt. Il consistait à faire l'ingénierie, le codage et l'intégration d'un module de navigation décrit dans la thèse de Fabien Flacher. Ce module de navigation a pour avantage d'être paramétrable et donc intéressant dans le cadre des expériences effectuées à l'AnimatLab. Ce projet ayant été réussi et validé par Olivier Sigaud certaines tâches m'ont été confiées dès mon arrivée au LIP6.

A. Description des objectifs

Dans un premier temps il m'a été demandé de compléter la documentation de GACS++ et d'intégrer le travail précédemment effectué dans le projet Kodabot. C'est une tâche qui a demandé beaucoup de temps car il n'était pas question de glisser des erreurs dans le tronc de Kodabot qui est utilisé par les développeurs de l'AnimatLab.
L'objectif principal est de mettre à disposition des chercheurs du LIP6 un jeu de contrôleurs de type GACS c'est-à-dire utilisant le framework de calcul développé et dont je connais le fonctionnement, ayant participé à sa réalisation. Ce type d'agents est totalement paramétrable et par conséquent devait être le résultat d'un algorithme génétique, comme expliqué dans la thèse originelle. En raison de certaines complications que nous verrons plus loin, ce paramétrage doit être fait à la main et pour faciliter ce travail une application doit être mise en place pour permettre le paramétrage d'un agent pendant que celui-ci tourne dans un simulateur donné.
C'est donc le but principal de mon stage qui s'inscrit dans un objectif commun « Obtenir des agents intelligents ». Une feuille de route a été mise en place dès le début, regroupant les objectifs de mon stage ainsi que ceux des autres membres de l'AnimatLab. Elle a permis d'observer la progression des différentes tâches.

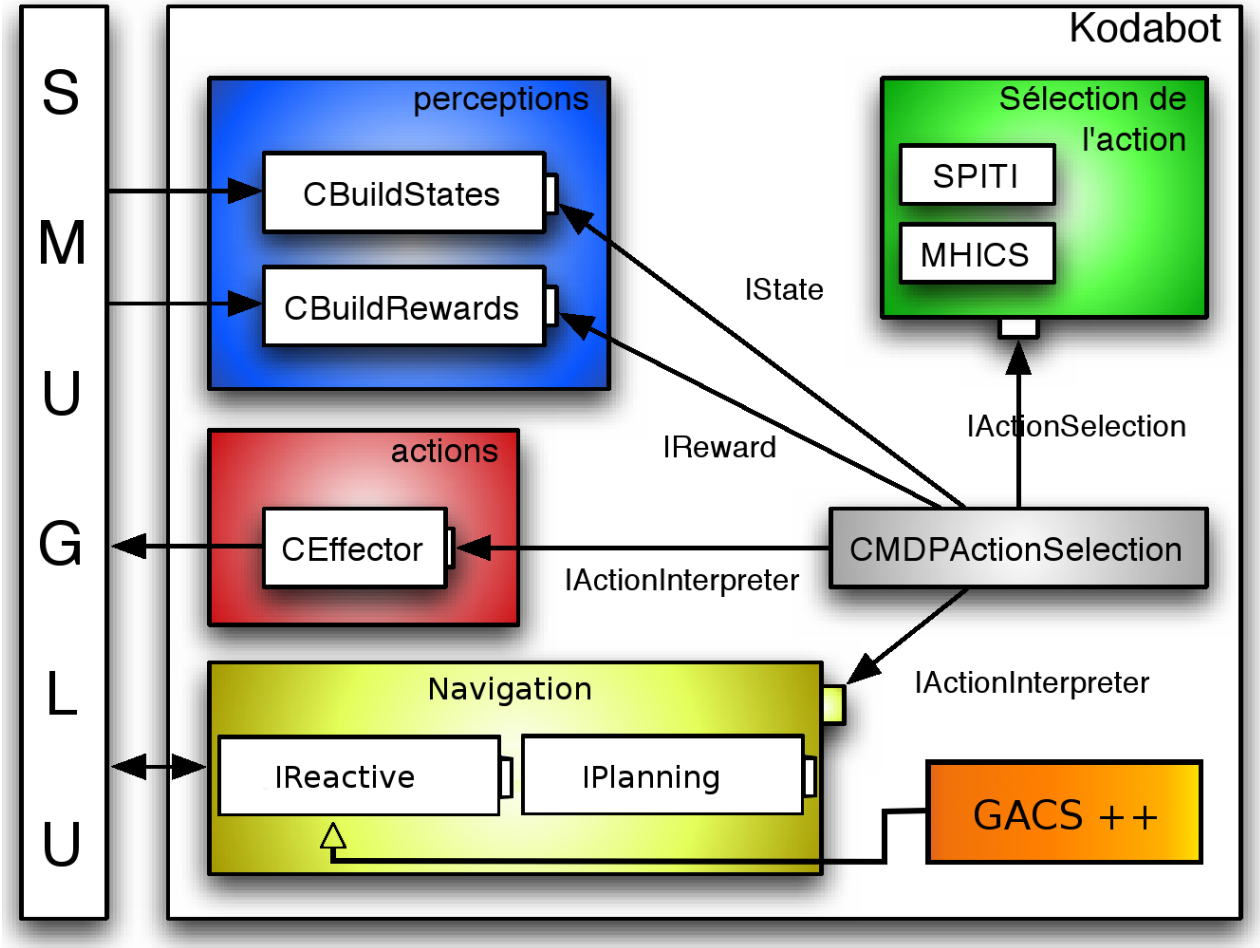
B. Apports scientifiques

L'AnimatLab travaille donc au développement d'agents intelligents, la navigation ne constituant qu'une partie du résultat à obtenir. Il est nécessaire d'avoir un contrôleur de type GACS pleinement opérationnel qui pourra être combiné avec des modules de planification, de sélection de l'action... Dans le cas de la planification, un travail de cartographie a été mené par d'autres stagiaires au LIP6. Ce module va apporter à GACS et vice-versa comme nous le verrons plus loin dans ce rapport. D'autre part Fabien Flacher va pouvoir utiliser l'application de paramétrage afin d'obtenir des contrôleurs pour le jeu vidéo *Counter-Strike* qui touche un grand public. C'est une étape de plus vers l'objectif principal qui est de confronter un agent du projet Kodamat de l'AnimatLab avec un agent scripté présent sur internet comme *PodBot* ou même des joueurs humains sur le Web.
En tâche de fond, il y a la volonté d'exposer et de mettre à plat les résultats de recherche obtenus à l'heure actuelle par l'AnimatLab.

Partie 2. Environnement de développement : Kodamat

A. Le projet Kodabot

SMUGLU est un ensemble d'interfaces développées à l'AnimatLab, il a pour but de pouvoir intégrer différentes parties d'un robot en un tout cohérent et exécutable, c'est-à-dire, de s'abstraire le plus possible du simulateur choisi pour mener les expériences. Le projet permet de connecter différentes parties de l'IA du robot entre elles, celles qui nous concerneront directement sont celle de navigation car GACS est un contrôleur de navigation.
Nous utiliserons aussi toutes les classes (interfaces) qui concernent les drivers. Ce framework est basé sur un gestionnaire de composants appelé SATOM.



Structure du projet Kodabot englobant SMUGLU.

L'architecture de Kodabot a été pensée dans le but d'être le plus modulaire possible. Cette modularité a été mise en oeuvre grâce à l'utilisation de composants SATOM. Cette architecture peut se diviser en deux grandes parties : le contrôleur et le simulateur. La charnière entre les deux est faite par l'ensemble d'interfaces SATOM que constitue SMUGLU. Les simulateurs sont également appelés drivers ou pilotes car les composants logiciels qu'ils implémentent ont généralement pour rôle de piloter un simulateur existant tel que Counter-Strike plutôt que d'en fournir un à partir de rien. Les simulateurs proposent un environnement virtuel dans lequel l'agent évolue. Les contrôleurs implémentent des composants logiciels dont le rôle est de contrôler l'agent dans un environnement. Le développement de différents contrôleurs a été initié

dans le cadre du projet Kodamat.

B. SATOM et SATOMJAVA

SATOM est un gestionnaire de composants, comme J2EE ou Corba, il est utilisé dans la plupart des développements de l'AnimatLab, et SMUGLU ainsi que Kodabot sont totalement dépendants de celui-ci. Cependant il est **en cours de développement et sans documentation**, ce qui ne simplifie pas la tâche des nouveaux arrivants sur des projets au sein de l'AnimatLab.
C'est un logiciel qui instancie et lie des composants entre eux pour créer un programme complexe. De plus, il est capable de gérer du code JAVA et C++. SATOM dispose également d'un système de comptage de référence pour gérer le cycle de vie des composants (détruire les objets inutilisés).
Pour fonctionner, le logiciel utilise des fichiers de configuration en XML :

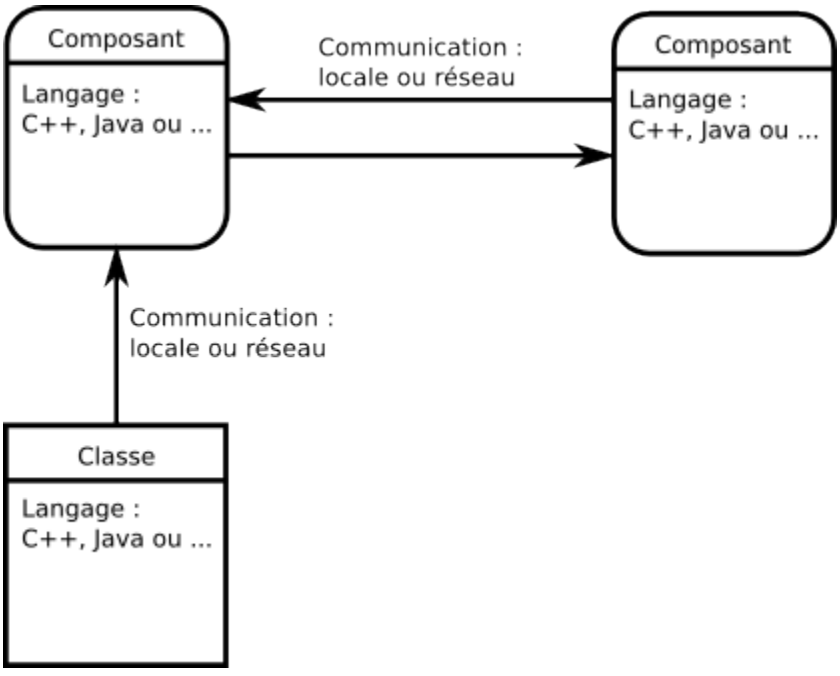
- des fichiers .stc dans lesquels doivent être déclarés les interfaces et composants d'un projet
- des fichiers .sata qui décrivent les interactions des composants

La politique de SATOM est pour résumer « what you get is what you did », car ce logiciel se contente de générer des fichiers. C'est à l'utilisateur de faire le travail nécessaire pour obtenir un composant utilisable, car le système est orienté interface et apporte donc très peu de services. L'utilisation des interfaces oblige à écrire le code qui va avec. Par exemple, même pour des interfaces très basiques comme l'utilisation de propriétés dans une classe, l'utilisateur devra gérer « à la main » l'implémentation des propriétés.

Ses principaux atouts sont :

- son chargeur de classes via fichiers XML (fichier SATA) qui permet d'éviter la recompilation des fichiers de tests
- la communication entre objets C++ et JAVA.

Ce dernier mécanisme permet actuellement, de composer un agent avec un algorithme de sélection de l'action écrit en JAVA nommé *SPITI* et développé par Thomas Degris. L'agent étant codé en C++ dans Kodabot.



La communication JAVA/C++ se fait de la manière suivante : à partir de JAVA l'appel à une

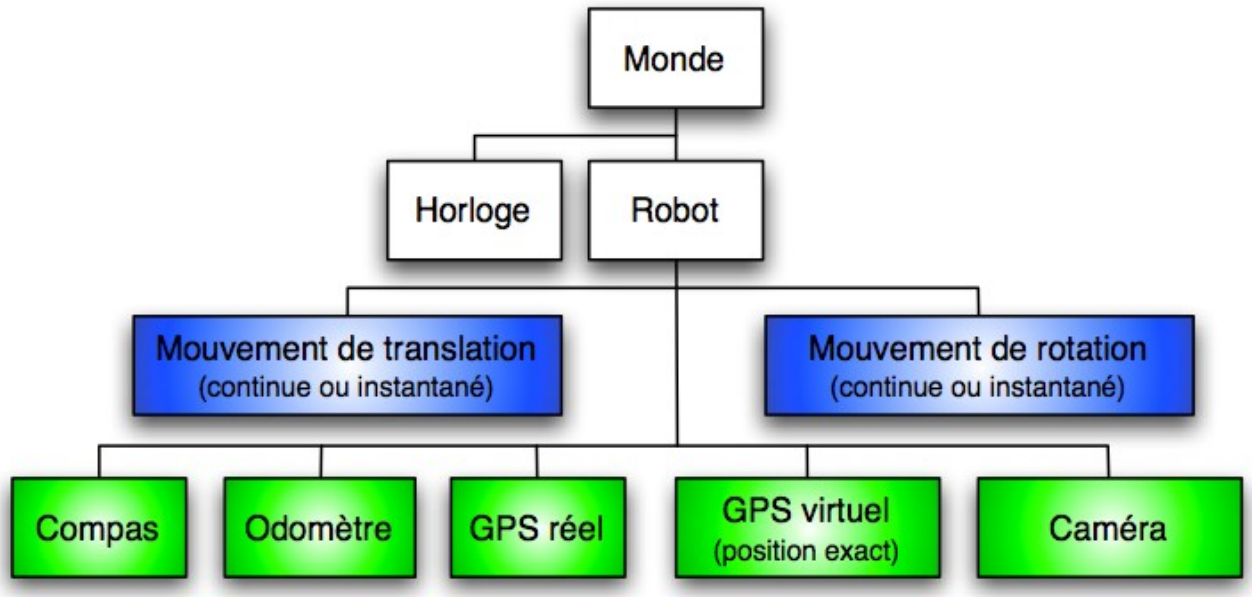
méthode d'une classe donnée est renvoyée vers la classe d'interprète associé, la « Gate ». Cet interprète a pour but de décoder le message. Ensuite l'interprète appelle une classe statique de SATOM qui, à partir de l'identité de l'appelant, va trouver l'interprète (autre que celui de JAVA) côté C++ capable de traiter le message.
Et, à partir de C++, c'est le même processus qui se produit : SATOM remarque que l'objet sur lequel on fait l'appel est local, il passe par la Gate pour transmettre le message qui est ensuite désérialisé et interprété; le retour est lui aussi serialisé dans une chaîne de caractère. Cette chaîne arrive à la Gate JAVA qui désérialise la réponse et la retourne. Il n'y a qu'un seul et unique interprète JAVA et une table de gates, la Gate en question est celle qui avait fait l'appel.

Le principe général est donc fondé sur l'envoi de messages traduisant les appels de méthodes. Le message est construit (sérialisation) puis interprété et traduit en un véritable appel (désérialisation). Ces chaînes de caractères respectent une syntaxe XML mais ce n'est en aucun cas une obligation, c'est le choix qui a été fait dans l'implantation de SATOM.

C. SMUGLU

SMUGLU, SiMUlateur GLUe, est une convention de l'AnimatLab définissant le protocole de communication entre un contrôleur et un simulateur. Il représente un compromis entre les différents besoins de chaque groupe de l'AnimatLab. Grâce à SMUGLU, il existe un réel découplage entre les algorithmes mis en œuvre au sein des contrôleurs d'une part et les composants formant les simulateurs d'autre part. Ainsi, la théorie voudrait qu'un contrôleur testé dans un simulateur A fonctionne également dans un simulateur B sans avoir à réécrire une seule ligne de code si ces deux simulateurs se conforment aux interfaces définies par SMUGLU. Le but ultime est de pouvoir charger le contrôleur dans un robot implémentant ces interfaces. En effet, ceci est intéressant car les expériences sur simulateur sont beaucoup moins onéreuses que celles sur un robot. Hélas, dans la pratique, le passage d'un contrôleur d'un simulateur à un autre n'est pas toujours aussi transparent et gratuit du point de vue du code. Par exemple, les principaux problèmes sont liés à des moteurs physiques ne réagissant pas de la même façon. Ainsi, il faut rendre le contrôleur robuste à toutes ces variations. Il faut noter également que tester son contrôleur dans deux simulateurs différents et évidemment bien plus coûteux que de le tester dans un seul. Donc, malgré l'utilisation de SMUGLU, un coût supplémentaire subsiste.

Structure globale



Représentation SMUGLU d'un simulateur

SMUGLU représente un simulateur sous la forme d'un arbre d'entités. Chaque entité est un composant implémentant l'interface IEntity permettant d'obtenir son type, son nom, la liste des entités filles, etc. Une entité peut également proposer des fonctionnalités supplémentaires adaptées à sa nature. Enfin, la racine de l'arbre est une entité proposant l'interface IWorldEntity. Sur la figure précédente, on distingue en bleu les effecteurs du robot et en vert ses capteurs.

Partie 3. La navigation dans Kodabot

Kodabot a pour rôle de regrouper les différentes parties constituant un agent, c'est-à-dire, la liste des récompenses pour les actions effectuées, la liste des états du bot, mais aussi le mécanisme de sélection de l'action ainsi que la planification et la navigation... Dans un jeu comme Counter-Strike, une bonne partie du problème des joueurs et des bots consiste à se trouver au bon endroit au bon moment. Résoudre efficacement ce problème suppose de savoir naviguer au sein de la carte sur laquelle se déroule le jeu. Et c'est le problème que nous allons aborder plus précisément dans cette partie.

A. Navigation *StraightToTarget*

Avant l'intégration de GACS au projet Kodamat il fallait bien disposer d'une navigation pour mener les expériences de sélection de l'action, cette navigation simpliste nommée *StraightToTarget* consiste à faire progresser l'agent en ligne droite jusqu'à atteindre une cible, cible qui sera mise à jour chaque fois qu'il atteint son objectif. Un problème évident se pose alors : comment choisir le prochain point? Mais aussi comment faire pour que le bot puisse l'atteindre en ligne droite?

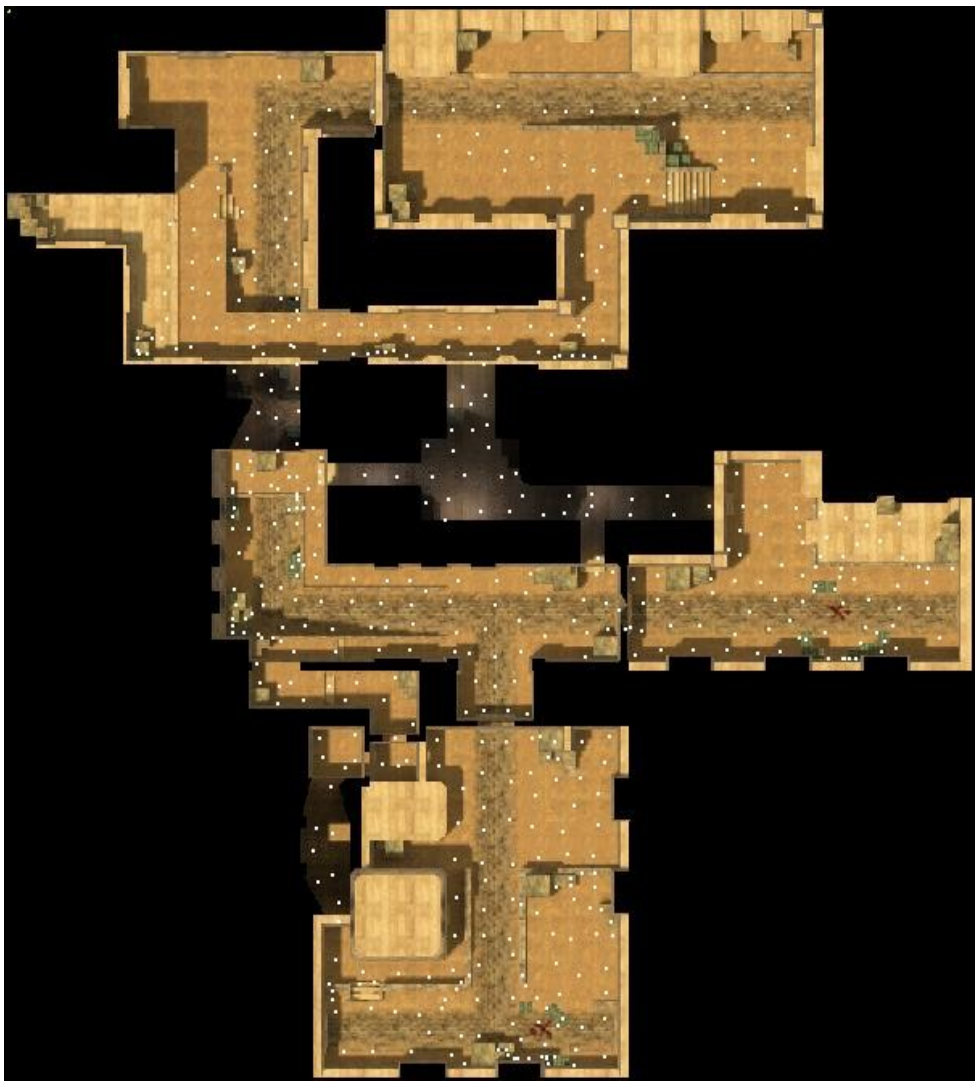
Pour naviguer, la plupart des bots "classiques" (par exemple, PodBot) utilisent un graphe de très nombreux points de passages ou *Waypoints* qui couvrent toute la carte en un réseau serré de telle façon que le bot puisse se rendre d'un waypoint au waypoint voisin en ligne droite sans rencontrer d'obstacle. Etant données la localisation courante et la destination du bot, il suffit de rechercher un chemin dans ce graphe à l'aide d'un algorithme comme A*, puis le bot n'a plus qu'à passer par la liste des waypoints qui constituent le chemin.

Problèmes rencontrés

Cette approche présente plusieurs inconvénients. D'une part, la construction du graphe de waypoints se fait à la main, ce qui est d'autant plus fastidieux que les waypoints sont très nombreux. D'autre part, la trajectoire du bot est très stéréotypée, puisqu'il emprunte toujours les mêmes lignes droites en passant par les mêmes waypoints successifs. En particulier, il n'est pas capable d'adapter cette trajectoire à la présence de partenaires ou d'adversaires autour de lui.

Pour ce type de navigation ce sont les graphes de PodBot qui sont utilisés, des graphes avec un maillage suffisamment serré, pour que le bot ne rencontre aucun obstacle entre 2 points donnés. L'inconvénient principal est la place mémoire utilisée pour stocker ce graphe ainsi que le temps de calcul pour calculer un chemin dans ce graphe. En effet si l'on prend par exemple la carte « de_dust » de Counter-Strike de la figure suivante sur laquelle sont affichés les sommets du graphes (pas les arêtes par soucis de clarté), voici ses caractéristiques :

- 500 sommets
- 2500 arêtes
- 3 à 5 arêtes par sommet



Carte « De dust » de Counter-Strike

B. Le principe de GACS

Le formalisme de GACS est l'un des résultats de la thèse de Fabien Flacher. L'idée consiste à construire le déplacement d'un agent comme résultant de la combinaison d'un ensemble de forces d'attraction et de répulsion qu'exercent sur lui des objets environnants. Plus précisément, le comportement d'un agent est sensible à un ensemble de points d'intérêt dont la localisation dépend des entités pertinentes présentes dans le domaine perçu par l'agent. Par exemple, dans le cas de Counter-Strike, ces objets pertinents sont :

- les partenaires
- les adversaires
- le point d'impact de rayons perceptifs sur les murs environnants
- certains sites particuliers comme le site de la bombe.

A chaque point d'intérêt est associée une force dont l'intensité varie en fonction de la distance entre l'agent et ce point pertinent, si bien qu'elle peut être attractive à une certaine distance et répulsive à une autre. Le déplacement de l'agent résulte d'une combinaison linéaire des forces qui s'exercent sur lui, chaque force se voyant attribuer un coefficient qui peut éventuellement varier en fonction de la situation.
Un point fort de GACS est que la définition d'un ensemble de forces exercées par un agent se ramène au bout du compte à un ensemble de paramètres numériques qui peuvent être réglés

pour optimiser le déplacement de l'agent.

C. Navigation GACS++

Les travaux menés sur la navigation réactive ont permis de résoudre les deux problèmes évoqués précédemment. D'une part, les bots sont capables de naviguer entre deux waypoints très distants l'un de l'autre sans suivre une ligne droite entre les deux, ce qui permet de réduire le nombre de waypoints nécessaires pour couvrir toute la carte. D'autre part, le formalisme de GACS décrit permet de mettre en oeuvre des capacités de navigation coordonnée qui font que des bots peuvent se déplacer en patrouille ou en formation et adapter leur trajectoire à la présence d'adversaires autour d'eux.
GACS++ n'est rien d'autre que le portage de GACS en C++. Un agent GACS++ est aux yeux de Kodamat un « IReactive », un composant qui contrôle le déplacement de façon réactive, c'est-à-dire qu'il répond à la commande *GoTo(position)* et se débrouille pour y aller. Il faut faire en sorte que les Waypoints soient un nouveau type de points d'intérêt qui seront pris en compte dans le calcul de GACS.

Composition

Nous allons nous arrêter sur la composition de cette navigation réactive, en général on y trouve les éléments suivants dans l'ordre dans lequel ils interviennent :

- Plusieurs *Detector* (un par type de point d'intérêt ou marque que l'on veut percevoir)
- Un *Sensor* gérant les portées de détection
- Un *Behaviour* constitué de plusieurs *Pattern* expliqué plus loin
- Un *Effector* qui réalise la normalisation

GACS permet le paramétrage d'un maximum de composants et ce comme expliqué dans la thèse de Fabien Flacher et plus précisément dans le papier suivant « GACS an Evolutionary Approach to the Spatial Coordination of Agents » par Fabien Flacher et Olivier Sigaud. Ces fichiers SATA sont assez pratiques car ils permettent de modifier quelques valeurs sans avoir à recompiler l'application. Les composants sont chargés au lancement; mais d'un autre côté le paramétrage a pour effet de demander beaucoup d'informations au niveau du SATA ce qui alourdit considérablement son écriture.

Dans le *Behaviour* on retrouve des composants paramétrables tels que des :

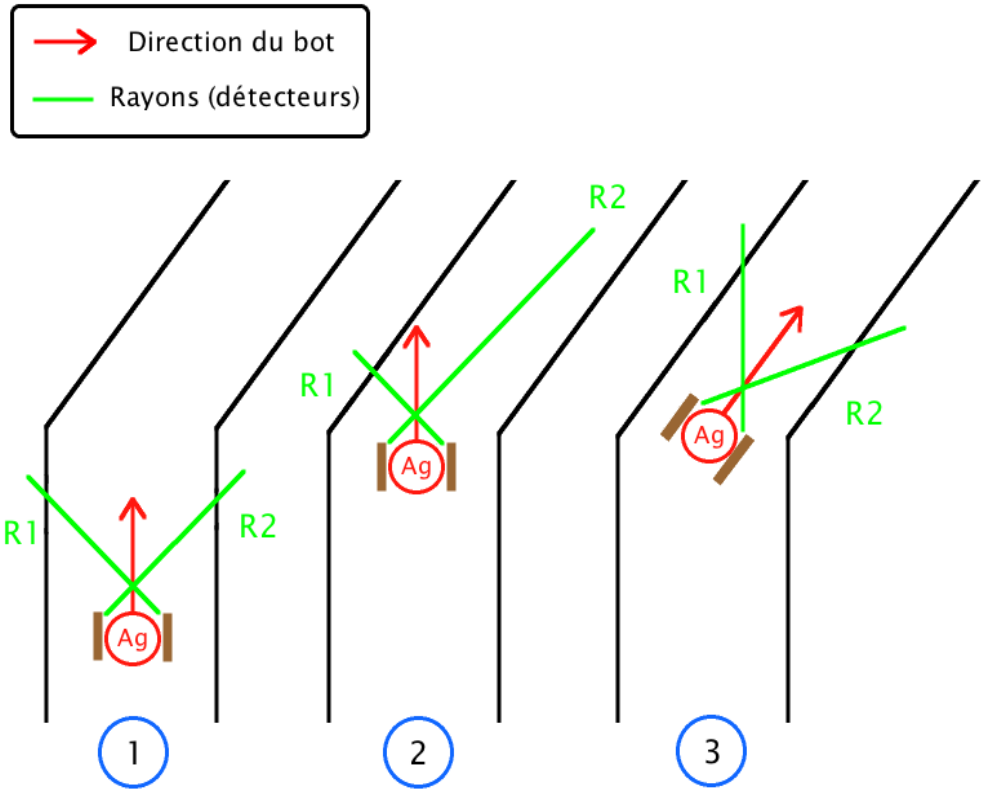
- fonctions d'orientation
- fonctions d'intensité
- fonctions de gain

Ce sont d'ailleurs les composants qu'il sera intéressant de paramétrer.

Exemple

Prenons un exemple simple : un agent de type *Braitenberg*. Dans un premier temps, nous allons expliquer rapidement le principe du *Braitenberg*, puis nous verrons la correspondance en terme d'agent GACS++.

Le principe de déplacement d'un agent de type *Braitenberg,* ou encore « véhicule de Braitenberg », est relativement simple. Il s'agit d'un agent muni de deux capteurs, en fait deux lancés de rayons (ici R1 et R2). Chacun de ses capteurs est relié à un frein associé à chaque roue. Plus la longueur du rayon est courte, c'est-à-dire, plus le point d'impact du rayon est proche, plus le frein se serre. Ainsi l'agent peut se déplacer en esquivant les murs et autres obstacles.



Fonctionnement d'un agent de type Braitenberg

Comment cet agent peut-il être représenté avec les composants de GACS++? Voici les constituants détaillés d'un tel agent.

Detector

Deux *Detector* vont être nécessaires, ici des *WallDetector* (un pour chaque lancé de rayon).

```
<component-data identity="animatlab.kodamat.kodabot.navigation.WALLDetector">
    <property name="generatedId" type="int" value="1"/>
    <property name="rayAngle" type="float" value="0.52359878"/>
</component-data>
```

Le même composant sera déclaré pour le rayon « droit » ayant, par exemple, un *generatedId* égal à 2 et un *rayAngle* égal à l'opposé de la valeur du rayon « gauche ». Les *id* donnés ici sont des identifiants de points d'intérêt.

Sensor

Un *Sensor* gérant les portées de détection pour chacun des deux rayons, par exemple les points d'impact pourront être visibles entre 0 et 100 mètres. Voici la gestion de portée pour le point d'impact du rayon « gauche » (*id* égal à 1).

```
<component-data
   identity="animatlab.kodamat.kodabot.navigation.CMapIdRangeEntry">
    <property name="id" type="int" value="1"/>
    <property name="closeRange" type="float" value="0.0"/>
    <property name="farRange" type="float" value="100.0"/>
</component-data>
```

Behaviour

Un *Behaviour* est une table mettant en relation un *Pattern* avec des fonctions d'orientation et d'intensité qui permettent de créer une force à partir des points trouvés. Le *Pattern* contient les *id* des points d'intérêt à traiter pour générer une force (ici 1 et 2, les points d'impacts des rayons). Et les fonctions précisent, quant à elles, l'*id* de la force résultante (ici 206).

```
<component-data
    identity="animatlab.kodamat.kodabot.navigation.CForceGenerationListEntry">


<!-- Le pattern contenant plusieurs entrées -->

<component-data identity="animatlab.kodamat.kodabot.navigation.CPattern"
                bind="pattern">
<component-data identity="animatlab.kodamat.kodabot.navigation.CPatternEntry">
    <property name="id" type="int" value="1"/>
    <property name="rank" type="int" value="1"/>
</component-data>

<component-data identity="animatlab.kodamat.kodabot.navigation.CPatternEntry">
    <property name="id" type="int" value="2"/>
    <property name="rank" type="int" value="1"/>
</component-data>

</component-data>


<!-- les fonctions de calcul de force -->

<component-data
   identity="animatlab.kodamat.kodabot.navigation.LinearIntensityFunction"
   bind="intensityFun">
    <property name="computedId" type="int" value="206"/>
    <property name="a" type="float" value="0.0"/>
    <property name="b" type="float" value="10.0"/>
</component-data>

<component-data
identity="animatlab.kodamat.kodabot.navigation.WallAvoidingOrientationFunction"
bind="orientationFun">
    <property name="computedId" type="int" value="206"/>
    <property name="angle" type="float" value="128"/>
    <property name="UTurnDistance" type="float" value="0.1"/>
</component-data>

</component-data>
```

Dans cet extrait de code XML on remarque que les fonctions de calcul de forces sont une fonction de calcul d'intensité linéaire ainsi qu'une fonction d'orientation d'évitement des murs. De même le Behaviour permet d'associer une fonction de gain à la force obtenue :

```
<component-data identity="animatlab.kodamat.kodabot.navigation.CMapIdGainEntry">
    <property name="id" type="int" value="206"/>
    <component-data
        identity="animatlab.kodamat.kodabot.navigation.ConstantGainFunction"
        bind="gainFun">
        <property name="constant" type="float" value="1.0"/>
    </component-data>
</component-data>
```

On retrouve bien la logique suivante : la force est créée avec un *id* égal à 206 et le gain doit

s'appliquer à cette force donc à son *id* (précisé en bleu).

Effector

Un *Effector* qui réalise la normalisation.

```
<component-data identity="animatlab.kodamat.kodabot.navigation.CGlobalEffector"
   bind="globalEffector">
    <component-data
      identity="animatlab.kodamat.kodabot.navigation.RegularNormalizeFunction"
      bind="normalizeFun">
        <property name="maximumForce" type="float" value="12.0"/>
    </component-data>
</component-data>
```

Ce qu'il faut retenir de cet exemple, c'est que malgré sa simplicité, il nécessite une description XML qui est fastidieuse à mettre en place. D'autant plus que je n'ai rappelé ici que les composants sans montrer tout le cadre dans lequel ils sont incorporés. Ainsi il est intéressant de se pencher sur une application de paramétrage et de génération automatique de fichiers de description d'agents en XML, dits fichiers SATA.

Partie 4. VisioGacs
Application de paramétrage en ligne

A. Motivations

Pourquoi mettre en place un outil de paramétrage en ligne d'un agent?
Tout simplement car GACS++, le module de navigation, est développé mais mal paramétré. Il est basé sur un système de calcul de forces permettant la configuration de la navigation de l'agent. Il vient compléter le module de planification qui, via un graphe de Waypoints (ou carte), fournit une cible vers laquelle se diriger. L'objectif est de remplacer la navigation de type *StraightToTarget* pour obtenir un graphe de Waypoints allégé ainsi qu'un déplacement plus souple et paramétré. On pourra de même obtenir différents comportements : seul avec et sans contournement d'agents, à plusieurs en formation (triangulaire, diamant ou « tas »).

Problème

Pour lancer l'application et effectuer différents tests, autrement dit pour instancier des objets dans GACS++, nous devons passer par l'outil mis en place par le gestionnaire de composants SATOM. Celui-ci nécessite de représenter les objets à créer au lancement dans un fichier SATA qui permet de décrire les composants souhaités ainsi que de les lier entre eux. L'intérêt d'une application de paramétrage est donc d'éviter d'avoir à stopper la simulation pour modifier le fichier de configuration « à la main » et ensuite relancer la simulation; ce qui est fastidieux.

Solutions

Pour répondre à ce problème il existe au moins deux solutions. La première consisterait en la programmation d'un algorithme génétique comme décrit dans le GACS d'origine (thèse de Fabien Flacher). Or pour lancer un tel algorithme il est nécessaire de pouvoir accélérer la simulation ce qui n'est pas possible dans le cas du jeu vidéo Counter-Strike dans lequel tous les tests sont effectués.
La seconde solution qui a été retenue, est l'utilisation d'une application tournant en parallèle de la simulation et qui permet à un utilisateur de paramétrer en « live » les composants de l'agent GACS. En effet un agent est totalement configuré dans un fichier au format XML (ou fichier SATA), fichier qui est chargé par le sérialiseur de SATOM. Le chargement est effectué une fois au lancement de la simulation, ce qui entraîne le redémarrage du simulateur à chaque modification manuelle faite dans le fichier SATA.

B. Mise en place

Dans un premier temps il a été nécessaire d'implanter des éléments inexistants dans SATOMJAVA et qui seraient utilisés dans l'application de paramétrage comme par exemple les accesseurs et modificateurs de champs C++ à partir de JAVA.
Il faudra pouvoir faire ce qui suit pour, par exemple, récupérer l'équipe d'un agent donné :

```
MobileAgentEntity agent...;

/* cast de l'entité en IProperties pour accéder à ces attributs */
IProperties agentProperty = (IProperties) Satom.stnew(agent, IProperties.class);
String team = agentProperty.getPropertyS("team");
```

Ce qui n'était pas implanté ici est la méthode « getPropetyS » mais il faut savoir qu'il y a une méthode différente pour chaque type de retour. Comme vu précédemment on récupérera la propriété d'un objet en faisant appel aux méthodes « getPropertyX » sur l'objet que l'on aura casté au préalable en l'interface « IProperties ». De même, il faut mettre en place une récupération des donnés complexes, c'est-à-dire, des bindings via l'interface « IBind ».

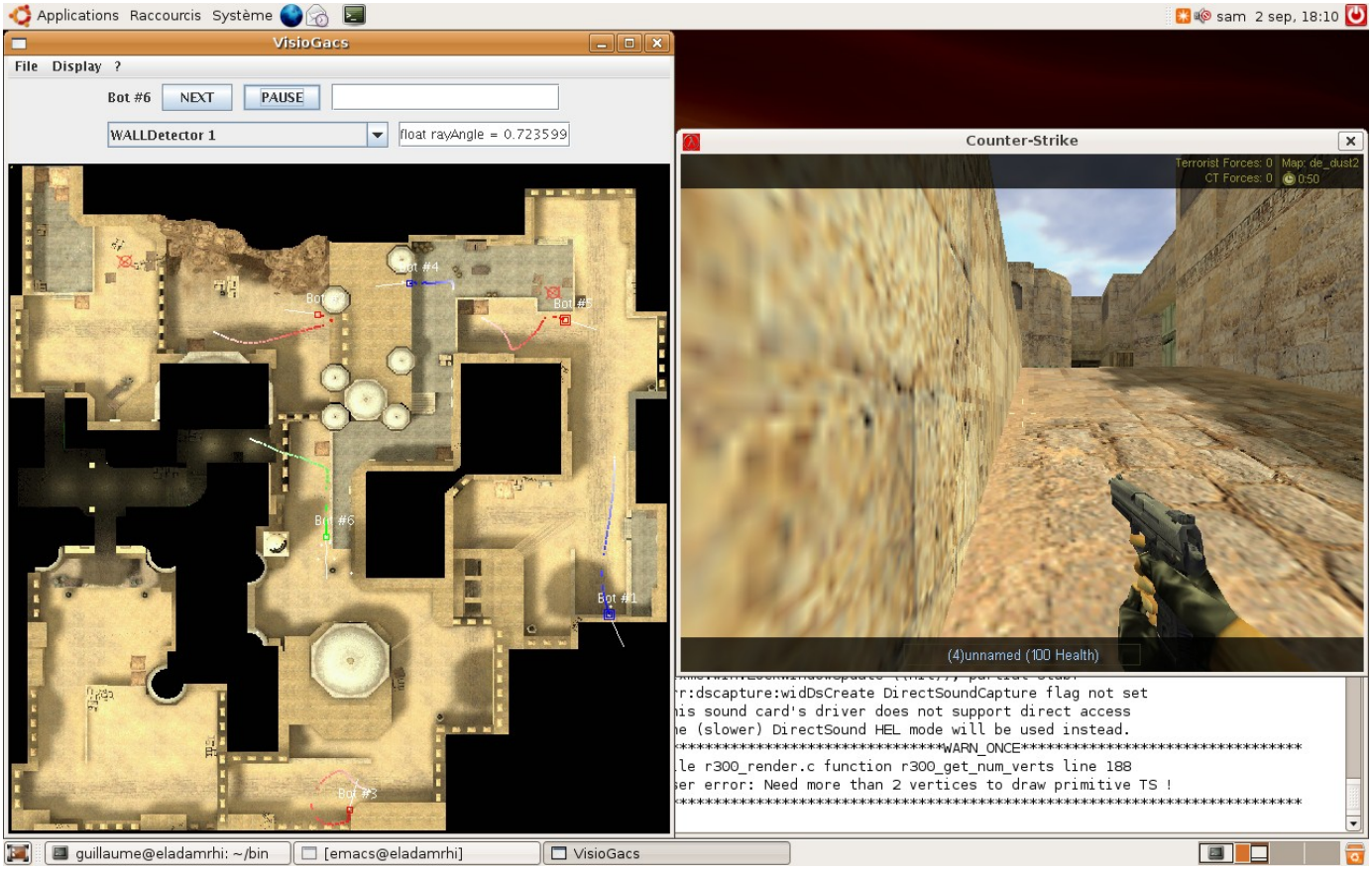
Pour implémenter la classe IProperties on s'appuie sur le mécanisme d'introspection de JAVA. Ainsi il faut implanter les méthodes « setProperty » et « getPropertyX » en utilisant le principe suivant :

1. Récupérer la classe de « this », l'objet courant
2. Récupérer les méthodes liées à cet objet, existe-t-il une méthode « getPropertyX » ou « setProperty » correspondante?
   Si oui on retourne le résultat de l'appel
3. Existe-t-il un *getter* ou un *setter* pour la propriété en question?
   Si oui on retourne le résultat de l'appel
4. Récupération des champs, existe-t-il un champ correspondant?
   Si oui on retourne le résultat de l'appel
5. Autrement une exception est levée car le champ est inexistant

Ces implantations se faisant directement dans SATOMJAVA elles ont donné lieu à de nombreux tests JUnit pour vérifier que la communication JAVA/C++ fonctionne correctement. On ne peut pas prendre le risque de distribuer des méthodes si bas niveau sans vérifier leur bon fonctionnement.

C. Structure de l'application

Pour mettre en place l'application de paramétrage qui accélèrera le développement d'agents réalistes, il a été décidé d'utiliser le langage JAVA. Ce qui peut paraître étonnant dans un premier temps, étant donné que les agents sont codés en C++. C'est une manière intéressante d'utiliser les propriétés de gestionnaire de composants utilisé au sein de l'AnimatLab, SATOM.



L'idée est de faire tourner VisioGacs en parallèle de la simulation pour pouvoir faire des réglages, comme le montre la figure précédente, avec à gauche l'application VisioGacs et à droite la simulation Counter-Strike.

Pour cela il va falloir charger une instance de VisioGacs en même temps que les agents à paramétrer. Ainsi, il va falloir passer par le fichier SATA contenant l'agent à observer et y ajouter un composant VisioGacs qui sera chargé. C'est de cette manière que VisioGacs s'intègre dans le reste de l'architecture Kodamat. Grâce à la gestion multi-langages, le sérialiseur XML de SATOM charge indifféremment en mémoire des instances C++ ou JAVA.

Prise en main

Grâce au projet effectué lors de la première année de Master j'avais déjà pris connaissance de la marche à suivre pour récupérer les informations nécessaires en C++; par contre je n'avais pas encore appris à le faire du côté Java. Cette prise en main a été plus longue que ce à quoi je m'attendais car il y a de nombreuses subtilités à comprendre et appliquer, comme par exemple les « cast » SATOM qui ne se font pas de la même manière que les « cast » JAVA.

En JAVA :

```
IWorldEntity world = (IWorldEntity) Satom.stnew("animatlab.smuglu.CWorldEntity",
                                                IworldEntity.class);
...

IPointerList entityList = this.world.getEmbeddedEntities();
IClockEntity clock = null;
for(int i=0; i<entityList.getSize(); i++) {
   if(entityList.get(i).toString().contains("CClockEntity"))
      clock = (IClockEntity) Satom.stnew(entityList.get(i), IClockEntity.class);
}
INotifier notifier = (INotifier) Satom.stnew(clock, INotifier.class);
notifier.addListener(this.getListener());
notifier.removeListener(this.getListener());
```

En C++ :

```
StRef<IWorldEntity> world("animatlab.smuglu.CWorldEntity");
...

StRef<IClockEntity> clock;
clock.Set(world->GetEmbeddedEntity("animatlab.smuglu.CClockEntity"));
IpointerList list = world->GetEmbeddedEntities(
                  "animatlab.kodamat.kodabot.navigation.CGACSReactive");
StRef<INotifier> notifier(clock);
notifier->AddListener(this);
notifier->RemoveListener(this);
```

Pour l'occasion une branche vierge a été mise à ma disposition sur Subversion pour accueillir l'application VisioGacs. Elle donne naissance à un nouveau composant dans le projet Kodamat, composant qui peut désormais être ajouté à n'importe quel fichier SATA permettant de contrôler les agents réactifs et de modifier les comportements des agents de type GACS; il est nommé : animatlab.kodamat.visiogacs.VisioGacs

La mise en place dans les fichiers SATA se fait en utilisant la technologie Xpath pour référencer des composants déjà définis dans le fichier :

```
<component-data identity= "animatlab.kodamat.visiogacs.VisioGacs">
      <component-data reference= "../../component-data[@bind = 'x']"/>
      [...autres bind et references sur des composants des agents]
<component-data/>
```

Dans cet exemple, le composant référencé est désigné grâce à un chemin relatif « ../../ » au noeud courant et à la valeur de son attribut « bind ». Cette manière de récupérer les éléments est très pratique car elle permet de créer une référence vers le graphe de *Waypoint* ou tout autre élément lié à l'agent et qu'il serait intéressant d'observer voire modifier.

Structure globale

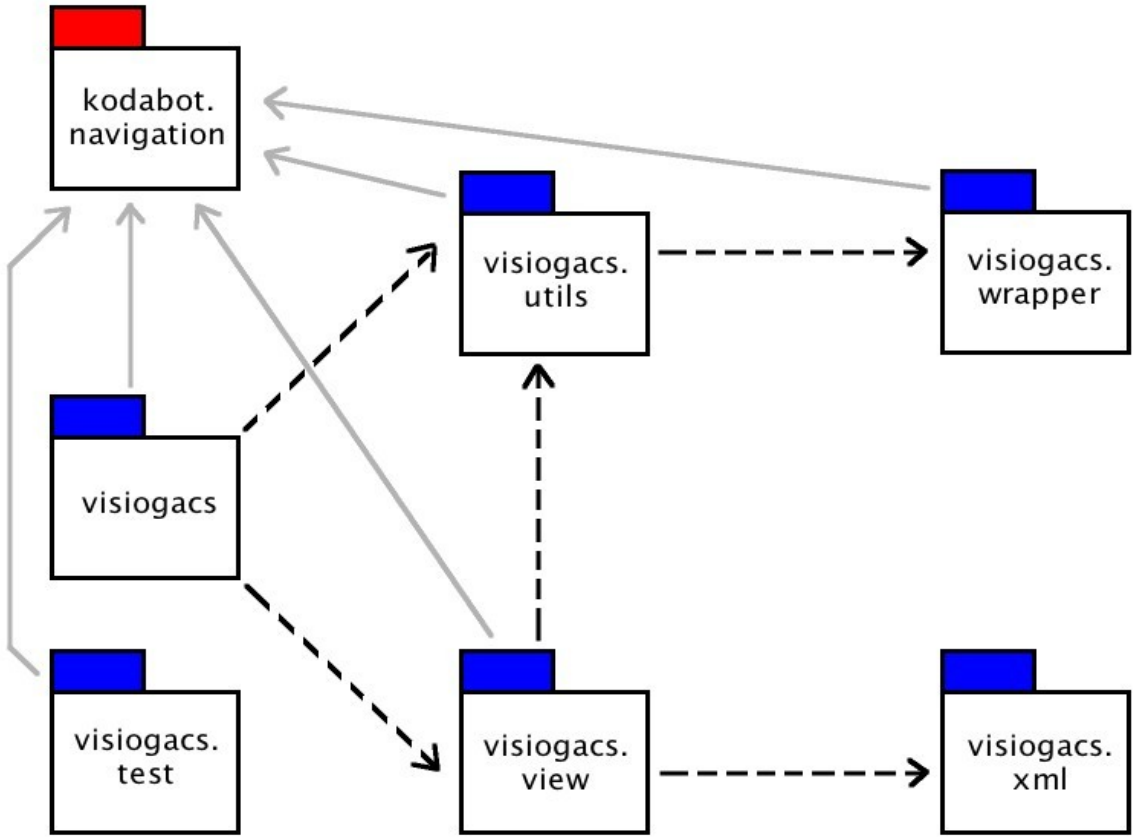


Diagramme de packages de VisioGacs

Les dépendances au package « animatlab.kodamat.kodabot.navigation » sont indiquées en gris; de la même manière l'ensemble des packages de VisioGacs dépendent du package « org.satom » ce qui n'est pas précisé sur le diagramme précédent.

A présent, passons à la description détaillée des différents packages composant VisioGacs.

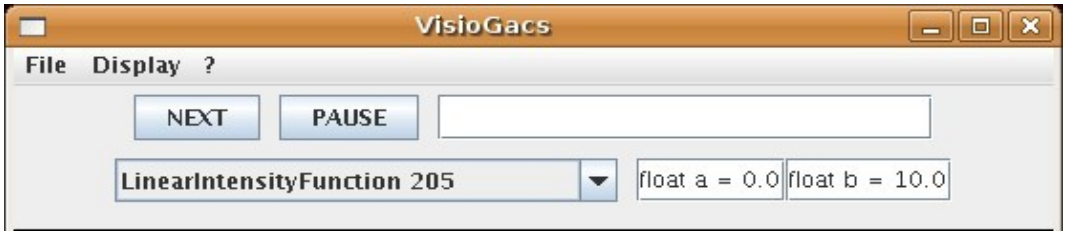
animatlab.kodamat.visiogacs

VisioGacs est la seule classe à ce niveau de package. Elle est chargée par le sérialiseur de SATOM, elle doit récupérer et initialiser les informations nécessaires par la suite dans le modèle mais d'autre part lancer la vue et le contrôleur.
Comme vu précédemment un agent peut-être vu sous « différents angles », c'est-à-dire, casté en différentes interfaces, chacune d'elles permettant l'accès à des méthodes particulières. Le travail de la classe VisioGacs réside principalement en la collection de ces différentes entités obtenues afin d'éviter des *cast* répétitifs et lourd en terme d'écriture, à travers le reste du code de

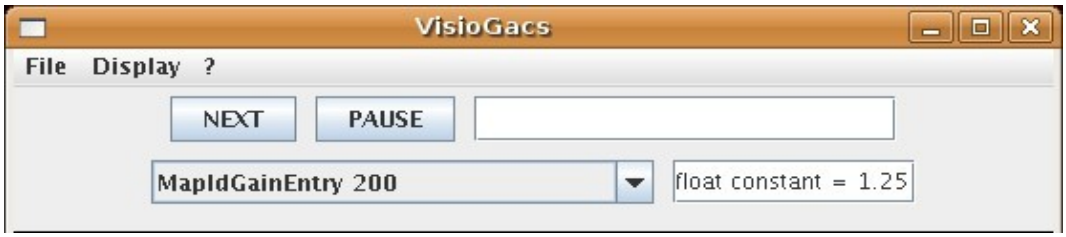
VisioGacs.
De plus si l'agent à charger dans VisioGacs est de type GACS alors on remonte l'ensemble de ces composants paramétrables en mémoire dans VisioGacs, ceci pour éviter les appels JAVA/C++ abusifs.

animatlab.kodamat.visiogacs.view

Ce *package* contient à la fois les parties vue et contrôleur qui sont toutefois clairement séparées de telle manière qu'il sera aisé d'ajouter de nouvelles fonctionnalités. VisioGacs sera très certainement amené à évoluer, ceci en fonction des besoins, si par exemple de nouveaux composants viennent s'ajouter dans GACS++. Le paramétrage et l'affichage 2D y sont gérés comme le montre les images qui suivent dans ce rapport. Affichage des agents, de leur points d'intérêts, des waypoints et gestionnaire de cartes.



Ces deux images représente le paneau supérieur de l'application VisioGacs. Il est possible de modifier l'affichage via le menu « Display », d'obtenir de l'aide (« ? »), de mettre l'agent en pause et d'en prendre le contrôle ou bien encore de le paramétrer en agissant directement sur les champs et les composants inclus dans la *ComboBox*.



On remarque que les valeurs paramétrables présentes dans le champs texte ont un identifiant ainsi qu'un type précis qui sont tous deux rappelés pour faciliter la prise en main. Par contre le nom des composants n'est pas très explicite ce qui force l'utilisateur à consulter le fichier SATA auparavant.

Des commandes simples ont été ajoutées comme le saut, le tir, le rechargement mais aussi le contrôle de l'agent : mise en pause et déplacement manuel... La pause est d'ailleurs la première des fonctionnalités qui a été disponible dans VisioGacs; elle était demandée par les différents participants au projet Kodamat au même titre que le contrôle manuel.

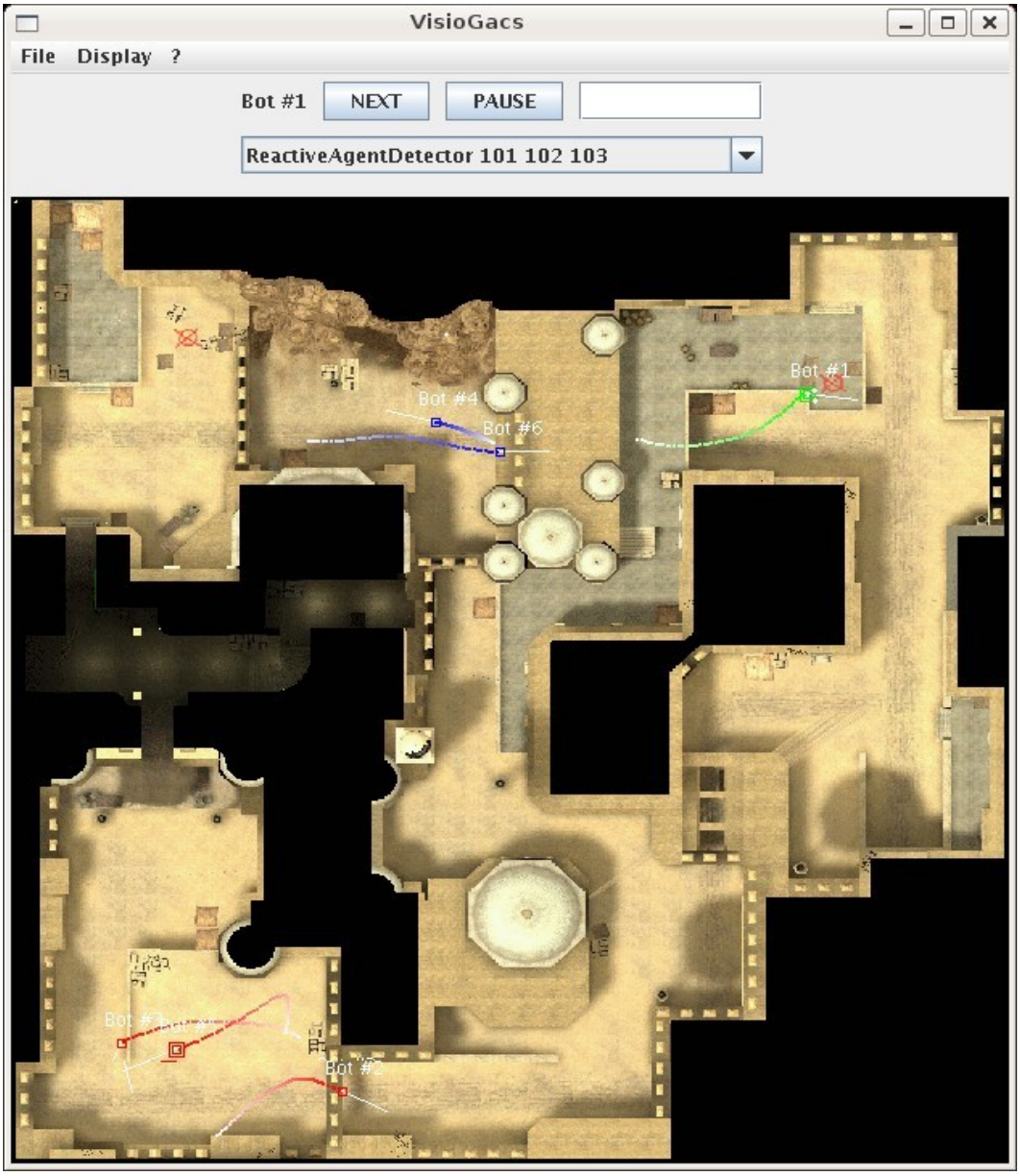
Affichage 2D des agents avec la carte courante utilisée sur le serveur en fond. Les proportions sont recalculées lors du changement de taille de la fenêtre. Il faut bien distinguer les trois dimensions que l'on manipule :

- la taille de l'image de fond
- la taille courante de la fenêtre
- la position de l'agent dans le repère général de la carte

Le système de gestion des cartes a été particulièrement soigné afin de pouvoir accepter d'autres cartes ce qui pourrait être intéressant à l'avenir, pour cela il faut :

- ajuster l'image de la carte au plus petit rectangle pouvant la contenir

- connaître la taille de l'image obtenue
- connaître les valeurs minimum et maximum de position de l'agent sur les axes X et Y



Six agents sur la carte « De dust 2 » vus par VisioGacs

Il y a un affichage des points d'intérêts qui est vraiment pratique. Quand l'agent est en pause les points sont mis à jour. Il faut alors demander à GACS de faire le calcul mais pas quand l'agent tourne sans quoi les calculs seront effectués en double (à chaque pas de temps) ce qui surcharge l'exécution inutilement.

De même on dispose de l'affichage de la direction courante de l'agent, de son nom et de ses derniers déplacements, mais aussi de son statut de leader et de son équipe (Contre-Terroristes en bleu, Terroriste en rouge et agent couramment sélectionné en vert). Cette partie nécessite des calculs trigonométriques car il faut bien noter que lorsque l'on fait un appel à getMoveSpeed(), le vecteur retourné est relatif au repère du bot, c'est-à-dire que les X sont croissants devant.

Le graphe de *Waypoint* dans VisioGacs peut-être affiché, cette fonctionnalité a été utilisée pour

afficher les sommets du graphe associé à la carte « De dust » de Counter-Strike (montrée précédemment dans ce rapport). Voici un exemple d'ajout intéressant que je vais détailler brèvement. Il est nécessaire de :

- dans le fichier SATA binder à « WaypointGraph » la référence au graphe
- dans le code effectif de VisioGacs charger l'ensemble de la structure pour éviter de répéter la communication JAVA/C++ inutilement

animatlab.kodamat.visiogacs.xml

Un parseur XML pour la sauvegarde de configuration après paramétrage a été mis en place. Dans ce package on notera l'utilisation de la technologie SAX. De plus certaines modifications ont été faites au niveau du sérialiseur de SATOM, c'est-à-dire, côté C++.

Le sérialiseur présent dans SATOM a 2 fonctions :

- Le chargement des fichiers SATA, l'instanciation d'objets décrit en XML
- L'écriture XML d'objets instanciés

Dans le cas présent c'est la deuxième fonctionnalité qui va être exploitée; ceci en récupérant comme suit une instance du sérialiseur puis en faisant un appel à la méthode Write() en mettant en paramètre l'objet à désérialiser :

```
StRef<IComponentSerializer> serializer("org.satom.CComponentSerializer");
serializer->Write(this);

virtual StVoid Write(IBase* component, StString documentPath);
```

Celle-ci permet l'écriture d'une instance mais pas des composants qui la constituent; il a donc été nécessaire de rentrer dans le code C++ de SATOM afin d'y ajouter les modifications nécessaires et indispensables comme l'écriture récursive des composants, c'est-à-dire, qu'il faut aussi sortir la description des composants constituant un composant donné; d'où la création d'une méthode :

```
WriteWithBindAttribute()
```

De cette manière le fichier XML récupéré en sortie n'était toujours pas satisfaisant dans le sens où il ne pouvait pas directement être chargé par le sérialiseur de SATOM. Il a fallu créer un « cadre » minimum dans lequel insérer le code XML de l'agent. Or ces modifications n'ont clairement rien à faire au niveau du sérialiseur. Ce n'est en fait pas à SATOM de s'adapter au besoin de VisioGacs mais plutôt à VisioGacs de faire le travail. En clair, il a fallu reprendre le fichier XML brut et le parser grâce à un parseur SAX qui se trouve être très adapté à ce genre de modification. Le SAX étant un parseur événementiel, on ne peut pas accéder à l'arborescence du fichier parsé mais l'ajout d'attributs, ou encore la modification de balises sont très faciles.

Avec ce package, la configuration courante de l'agent peut-être sauvegardée à tout moment.

animatlab.kodamat.visiogacs.wrapper

Dans ce package on retrouve la définition de *Wrapper* d'objets composant le bot, c'est-à-dire, d'objets C++ non récupérables directement en JAVA. Notamment les Map C++ utilisées du côté de la navigation (package animatlab.kodamat.kodabot.navigation) dont les entrées ne sont pas récupérables en JAVA.

Par exemple, prenons les tables mettant en relation un *Pattern* et des fonctions de calcul de forces, tables utilisées pour représenter le composant *ForceGenerationList* en mémoire. Lorsque l'on souhaite obtenir une entrée de cette table le mécanisme est le suivant :

- Création d'un *Wrapper* JAVA de VisioGacs
- On donne à cet objet une référence à la Map complète

– On indique à cet objet l'indice de l'entrée que l'on souhaite récupérer

Ce système ne pose pas de problème particulier étant donné que la table complète est passée par référence à l'objet *Wrapper*.

animatlab.kodamat.visiogacs.utils

Classes constituant la majeure partie du modèle, ces classes regroupent les informations utilisées par la vue; voici la liste des parties du modèle :

– Gestion de l'agent : Agent, GacsAgent & AgentList
– Gestion des composants GACS : InterestingComponent & InterestingComponentList
– Gestion du graphe de *Waypoint* : WaypointGraph & Waypoint
– Gestion d'objets supplémentaires : Positions & Utils

Par exemple la classe Agent conserve les multiples casts de l'agent en diverses interfaces de SMUGLU, comme vu précédemment. La classe GacsAgent hérite d'Agent et a pour rôle de charger les composants GACS.

animatlab.kodamat.visiogacs.test

Ce package contient une batterie de tests JUnit (à enrichir faute de temps) qui a été nécessaire pour tester le bon fonctionnement des premières communications JAVA/C++. En effet cet aspect n'a pas été évident à prendre en main. Par exemple on y retrouvera des tests simples comme :

```
/**
 *
 * test1c : Another modification of a StringList
 *
 *
 **/
public void test1c() {
   this.stringList = (IStringList) Satom.stnew("org.satom.CStringList",
                                                IStringList.class);
   this.stringList.pushBack("I'll");
   this.stringList.pushBack("be");
   this.stringList.pushBack("back");
   this.stringList.pushBack("!!");
   assertEquals(this.stringList.get(2), "back");
}

/**
 *
 * test2a : Creation of a FloatList
 *
 *
 **/
public void test2a() {
   this.floatList = (IFloatList) Satom.stnew("org.satom.CFloatList",
                                                IFloatList.class);
   assertNotNull(this.floatList);
}
```

Partie 5. Paramétrage et résultats obtenus

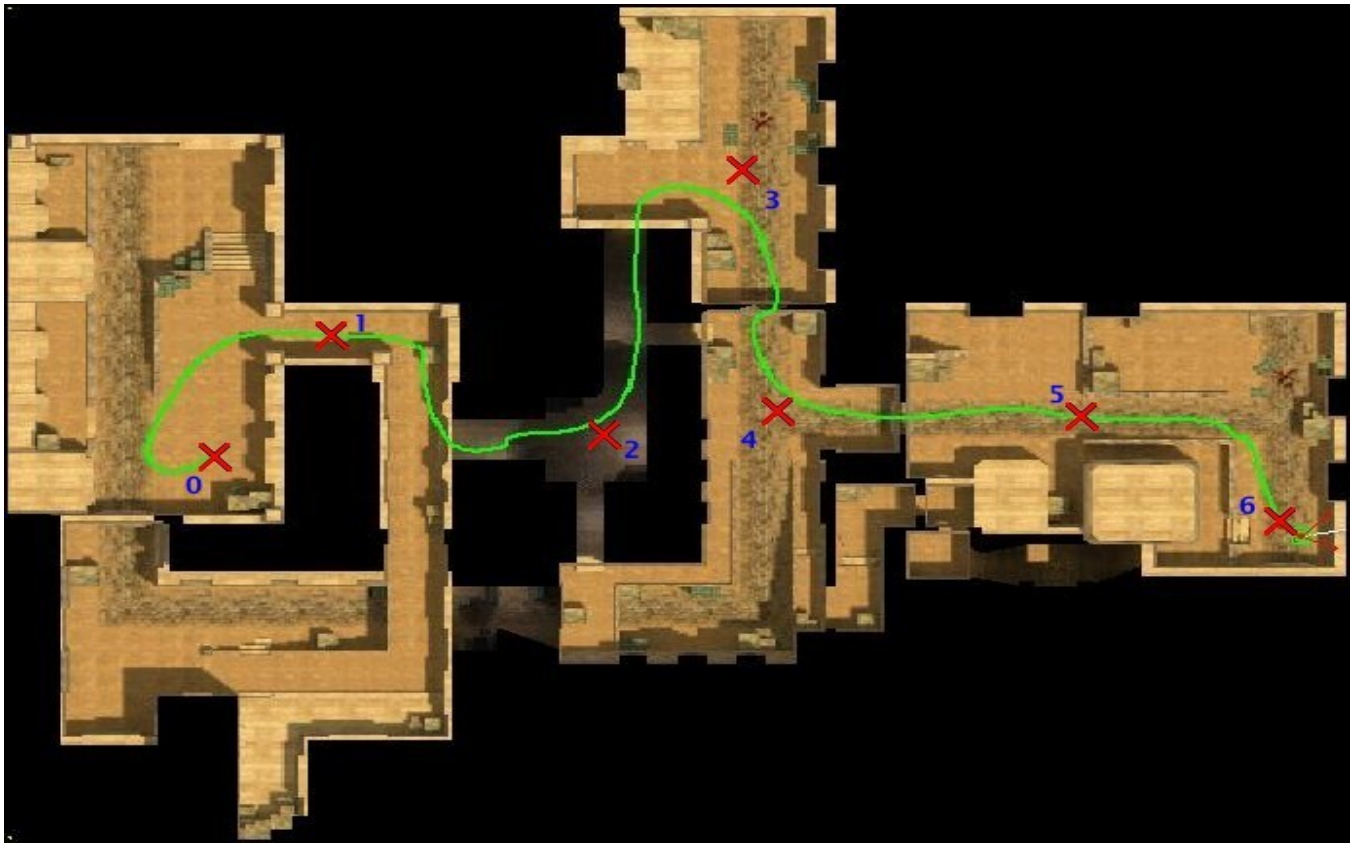
L'après VisioGacs n'est que le commencement... en effet l'objectif final n'est pas de créer l'application de paramétrage mais de l'utiliser pour avoir différentes configurations d'agents réalistes :

- Un bot de type Braitenberg ayant un objectif
- Contournement d'agents
- Formation triangulaire

A. Force vers le but et contournement d'obstacles

Une fois que l'on a obtenu un agent de type *Braitenberg* bien paramétré, comme vu précédemment, l'aspect contournement d'obstacles est géré. Or on souhaite ajouter une force d'attirance vers le but puis observer les résultats. La force vers un but est très simple à mettre en place, car elle est déjà implantée dans GACS++, il suffit pour cela de fixer la cible courante de l'agent. Une fonctionnalité intéressante a été mise en place dans VisioGacs pour exploiter cette propriété, ainsi en faisant un clic de souris sur la carte (dans la fenêtre de VisioGacs), la cible de l'agent courant est directement remise à jour avec les nouvelles coordonnées.

Avec un tel dispositif, il a été possible de paramétrer l'agent pour équilibrer les forces d'attirance au but et celles de répulsion aux obstacles. Le meilleur résultat obtenu est que l'agent suit le chemin désiré en ne passant que par un *Waypoint* sur quatre! Par contre en « guidant » le bot à la souris, on peut encore diminuer l'utilisation de *Waypoint*. Je pense qu'en redisposant les *Waypoint* judicieusement sur la carte il serait possible de diviser leur nombre actuel par dix. Le principal avantage est la diminution significative du nombre de sommets et d'arêtes du graphe, induisant un chargement plus rapide de la simulation et un calcul de chemin bien plus efficace.

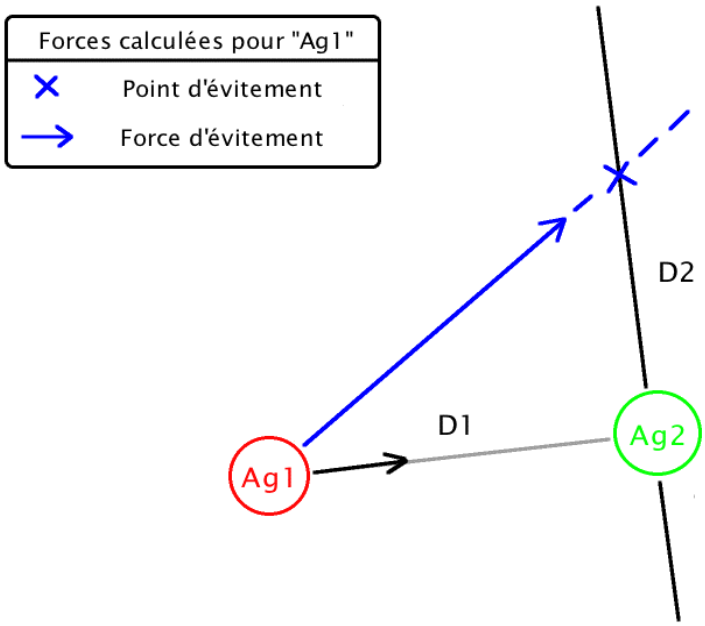


Dans le cliché précédent on remarque que l'on peut faire traverser la carte à un bot de ce type en six clics de souris, indiqués par des croix rouges, chaque clic remettant à jour la cible à atteindre et l'indice « 0 » étant la position initiale de l'agent. Cette situation ne serait pas possible avec un bot de type *StraightToTarget* vu précédemment, en effet, dans l'exemple les cibles sont séparées par des obstacles si bien que l'on ne peut pas atteindre la suivante en ligne droite. Avant, près d'une quarantaine de *Waypoint* étaient nécessaires pour réaliser ce parcours.

B. Evitement des autres entités

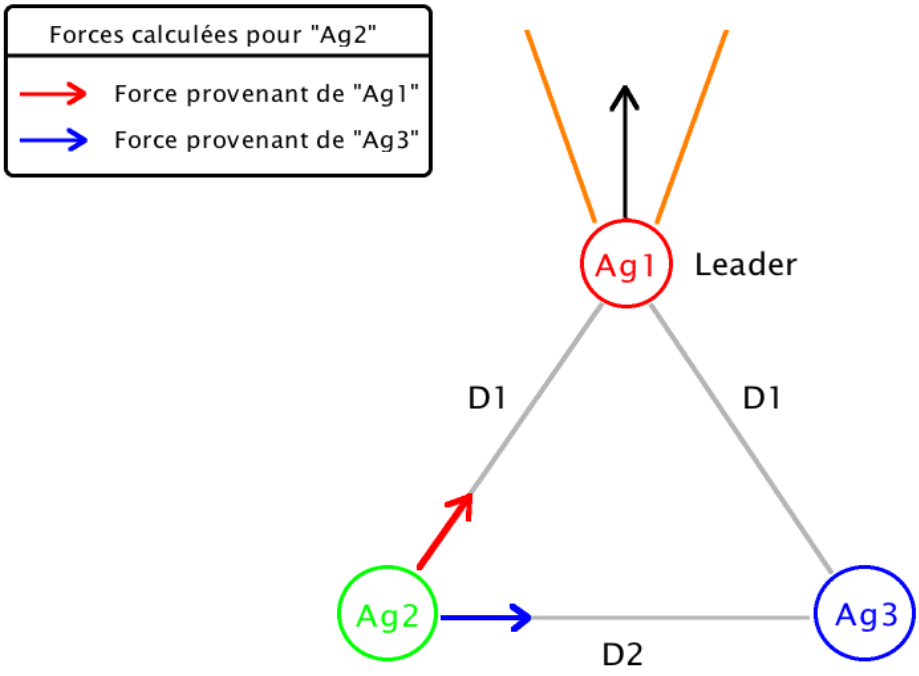
Ces deux images provenant de VisioGacs montrent l'importance de pouvoir éviter les entités présentes sur la carte. En effet, si l'on met en scène 2 agents GACS ayant un comportement de type *Braitenberg*, on obtient le cas présenté par l'image de gauche, c'est-à-dire que les 2 agents de ne se voient pas et finissent l'un contre l'autre.
Par contre, sur l'image de droite, les 2 agents sont équipés de détecteurs qui leur permettent de percevoir les entités environnantes telles que les amis, les ennemis et les leaders d'équipe. Dans ce cas l'impact est évité grâce à une force de répulsion.

Voici le principe de l'évitement : il faut avant tout pouvoir détecter les entités environnantes à une certaine distance D1, puis le calcul se fait en deux temps. D'abord un point d'évitement est calculé « à côté » de l'agent que l'on souhaite contourner; ici à gauche à une distance D2. Il suffit ensuite d'appliquer une force vers le point généré ce qui va avoir pour effet de dévier la résultante du calcul de GACS vers la gauche.



En théorie cela paraît être très efficace, seulement ça ne fonctionne vraiment bien dans la pratique que sur quelques cas simples comme montré ci-dessus. C'est-à-dire lorsque l'on fait se rencontrer deux agents qui arrivent l'un en face de l'autre. Autrement, dans la plupart des cas, soit la force d'évitement a une incidence trop forte sur le résultat auquel cas l'agent tente d'éviter les autres quitte à rencontrer un obstacle; soit la force d'évitement est trop faible pour éviter la rencontre. J'ai tenté en vain d'obtenir un paramétrage correct via VisioGacs et pour avoir passé un certains temps sur ce problème, je pense que le paramétrage est très difficile à optimiser dans des cas aussi complexes que celui-ci.

C. Maintien de formation



Pour le maintien de formation, il a fallu se pencher sur la modélisation du système de force. Comment doit-on agencer les composants pour obtenir une formation triangulaire? C'est-à-dire, quels composants utiliser, et ceci avant même de se poser les questions de paramétrage.

Deux systèmes ont été pensés. Le premier est un modèle avec leader, représenté sur la figure suivante. Il repose sur un leader de type *Braitenberg* et ayant une force vers un but; il servira de « guide ». Les deux autres agents, quant à eux, sont « aveugles » et ne perçoivent que leur alliés via des capteurs d'entités. Prenons l'exemple de l'agent Ag2, deux forces lui seront appliquées, la première le tenant à une distance D1 du leader Ag1, la deuxième à une distance D2 de son homologue droit Ag3.

Le second système vient en réponse à la question suivante : que se passe-t-il si le leader vient à disparaître? En effet dans le cas précédent si le leader disparaissait, les deux autres agents seraient perdus. Pour résoudre ce problème le second système part du principe qu'il n'y a pas de leader dans une équipe; tous les agents d'une même équipe sont alliés et ont le même comportement. Pour cela chaque agent à sa propre attirance vers un but ainsi que son évitement d'obstacles. Le problème d'un tel système est encore une fois le paramétrage, c'est-à-dire la proportion à donner entre les forces de maintien de formation, d'évitement d'obstacles et de l'objectif. A chaque fois que l'on ajoute une nouvelle notion, et donc au moins une nouvelle force qui intervient dans le calcul, il faut revoir l'ensemble des paramétrages.

Conclusion

Revenons point par point sur les résultats obtenus. Tout d'abord l'application VisioGacs, fonctionne et permet un paramétrage plus rapide, seulement elle ne fait qu'alléger cette opération qui est pour le moins laborieuse. Elle n'apporte pas la solution tant attendue, c'est-à-dire un paramétrage facile et universel. Les résultats que j'obtiens, même s'ils sont intéressants, sont bien moins importants que prévus au tout début de mon stage. La navigation présente des problèmes qui interviennent dans certains cas imprévus. En effet, on peut passer un temps considérable à configurer un agent sur un parcours d'une carte donnée, être satisfait du résultat obtenu et se rendre compte en testant ce bot sur une zone différente ou dans un cas bien précis que ça ne correspond pas du tout à l'attente; le bot se bloquant dans un coin de la carte. L'enjeu était de porter la navigation GACS dans les expériences de sélection de l'action réalisées par Thomas Degris, Nicolas Desprès et Jean-Philippe Dubus; qui utilisent une navigation de type *StraightToTarget* limitée et présentant de nombreux inconvénients. La réponse à ce problème peut être donnée par une utilisation de VisioGacs par un expert dans le domaine, en tout cas VisioGacs est facilement modifiable et améliorable.
Durant ce stage j'aurais appris à monter une véritable application JAVA, reposant sur un modèle MVC, non pas fermée sur elle-même mais devant communiquer avec divers systèmes en développement et partiellement documentés comme SATOM, SMUGLU et Kodabot. J'ai dû mener ce projet seul, et ainsi gérer un répertoire sous Subversion vierge de tous fichiers au lancement de mon stage. Cela m'a demandé de mettre en place un structuration claire des sources ainsi que les fichiers permettant aux utilisateurs d'installer facilement ce composant.
Je remercie encore une fois les personnes m'ayant aidé durant ce stage qui a été très enrichissant et qui me rapproche petit à petit de mon objectif professionnel.

« Si, réduits au désespoir, ils viennent pour vaincre ou pour périr, évitez leur rencontre »,
Sun Tzu, L'art de la guerre, Article VII.

Lexique

GACS

Génération Ascendante de Coordination Spatiale, thèse de Fabien Flacher.

Counter-Strike

C'est le simulateur utilisé dans le cadre des recherches effectuées par l'AnimatLab. C'est en fait un jeu de type FPS (First Person Shooter) connu des joueurs et très joué sur le Web. Il est utilisé car c'est celui pour lequel les drivers ont été le plus développés pour récupérer les informations nécessaires au calcul des déplacements mais aussi pour la partie, sélection de l'action.

PodBot

Équipe de bots scriptés pour Counter-Strike développé par *CountFloyd*. L'IA employée est du niveau d'un joueur moyen. C'est la référence actuelle en la matière.

SPITI

SPITI est le système d'apprentissage par renforcement et de sélection de l'action développé par Thomas Degris dans le cadre de sa thèse.

Annexe

A. Manuel d'utilisateur de VisioGacs

Install VisioGacs application :

Do a 'make install' of SatomJava after Satom, Smuglu and Kodabot are installed.

Then use the build.xml file by making :
ant -Dsatom=$SATOM satomInstall

Launch the VisioGacs application :

Choose a Sata file and add the following lines :

1) Labels need to be added on bots we want to control : label="botx" as followed :

```
<component-data identity="animatlab.kodamat.kodabot.navigation.CGACSReactive"
label="bot1">
  [...]
</component-data>

<component-data identity="animatlab.kodamat.kodabot.navigation.CGACSReactive"
label="bot2">
  [...]
</component-data>

[...]
```

2) Then use these labels by declaring only one VisioGacs component which contains all references :

```
<component-data identity="animatlab.kodamat.visiogacs.VisioGacs">
  <component-data reference="/satom-data/component-data[@label = 'bot1']"/>
  <component-data reference="/satom-data/component-data[@label = 'bot2']"/>
  [...]
</component-data>
```

Control commands :

Panel's buttons :
NEXT - switch to the next bot
PAUSE - stop the current bot

In controller text field :
UP/DOWN & LEFT/RIGHT - move the bot
SPACE - make the bot jump
CTRL - make the bot crouch
R - make the bot reload
S - make the bot shoot
X - make the bot change shoot

In parameter text field :
W/X - down/upgrade of 0.05 value contained in the current field

Q/S - down/upgrade of 1.0 value contained in the current field
A/Z - down/upgrade of 100.0 value contained in the current field

Basic text output, XML output and screenshot :

This function is accessible from the menu bar in File -> Save, the result is a text output which looks like that :

```
Name = Bot #2

WALLDetector 1 (
   float rayAngle = 0.5235988
)

[...]

LinearIntensityFunction 206 (
   float a = 0.0
   float b = 10.0
)
WallAvoidingOrientationFunction 206 (
   float angle = 64.0
   float UTurnDistance = 0.1
)

[...]

ConstantGainFunction 206 (
   float constant = 1.0
)

[...]
```

and also an XML output that can be directly loaded, and a screenshot corresponding to drawing panel content.

B. Capture de vidéos dans Counter-Strike

Voici le script qui m'a permis de faire des vidéos pour le site Web de l'AnimatLab. Il sert notamment de *HOWTO* pour la capture de vidéos dans Counter-Strike, mais il permet aussi d'automatiser les tâches de conversion et la création de la vidéo finale. Au préalable il sera nécessaire de fixer la variable d'environnement nommée « HALF_LIFE » qui pointe sur le répertoire de Half-Life, dans mon cas sa valeur est : $HOME/c_drive/Sierra/Half-Life/

```
#!/bin/sh

# Debut de convert.sh

FILE=$1

# How to get bmp images from CStrike?
# 1/ Launch a server and a client
# 2/ In client's console do :
#    a- "record X" (where "X" is the name of ".dem" file)
#    b- "stop" and quit the client
# 3/ Launch a client, open a console and do :
#    a- "timedemo X"
```

```
#    b- "startmovie Y Z" (where "Y" is the name of ".bmp" files and "Z" the
number of frame per second, 25 is a good value)
#    c- "endmovie"

# Import bmp images from Half-Life directory
mkdir $1
mv $HALF_LIFE/$1*.bmp $1/

cd $1

# Convert bmp images to jpg format
ls $1*.bmp | xargs -i basename \{\} .bmp | xargs -i convert -verbose \{\}.bmp
\{\}.jpg

# Remove all bmp images
rm -f $1*.bmp

# Make an avi video from jpg images
mencoder mf://$1*.jpg -mf w=800:h=600:fps=15:type=jpg -ovc lavc -lavcopts
vcodec=mpeg4 -oac copy -o $1.avi

# Make an mpeg video from an avi video
#mencoder -ovc lavc -oac copy $1.avi -o $1.mpg

# Read the video obtained
mplayer $1.avi

cd ..

# Fin de convert.sh
```